Abrégé de programmation en Langage C

Denis Robilliard

version 2010

Laboratoire d'Informatique du Littoral
Université du Littoral-Côte d'Opale

Table des matières

I Introduction **2**
1 Origines . . . 2
2 Différences principales avec Ada . . . 3
3 Quelques rappels de base . . . 3
3.i Compilation . . . 3
3.ii Gestion de la mémoire . . . 3

II Eléments de base **4**
1 Notions élémentaires . . . 4
1.i fonction `main`, instructions, bloc, expressions élémentaires . . . 4
1.ii Commentaires . . . 4
2 Variables, littéraux et types simples . . . 4
2.i Les caractères . . . 5
2.ii Les entiers . . . 5
2.iii Les réels . . . 6
2.iv Les booléens . . . 6
2.v Les littéraux chaînes . . . 6
2.vi Le pré-processeur . . . 6
2.vii Entrées/sorties au terminal . . . 7
3 Exemple de programme source . . . 7

III Contrôle du flot d'instructions **8**
1 Expression du choix : alternative et aiguillage . . . 8
1.i Alternative — if . . . 8
1.ii Aiguillage — switch . . . 8
2 Boucles . . . 9
2.i Tant que — while . . . 9
2.ii Pour — for . . . 9
2.iii Faire tant que — do while . . . 9
2.iv Compléments sur les boucles . . . 10
3 Sauts et labels . . . 10

IV Expressions **10**
1 Définition . . . 10
2 Opérateurs (listés par priorité décroissante) . . . 11

V Pointeurs, tableaux statiques à 1 dimension, chaînes de caractères **12**
1 Pointeurs . . . 12
2 Tableaux statiques à 1 dimension . . . 12
3 Lien entre pointeurs et tableaux. . . . 13
4 Variables chaînes de caractères . . . 13

VI Fonctions **15**
1 Syntaxe . . . 15
2 Quelques exemples de fonctions . . . 16
3 Simuler les paramètres en mode "in out" . . . 16
4 Les macros du pré-processeur . . . 17

VII Tableaux dynamiques, tableaux à 2 dimensions **18**
1 Tableaux dynamiques . . . 18
2 Tableaux à 2 dimensions . . . 19

VIII Structures enregistrements, définition de nom de types **20**
1 Structures enregistrements . . . 20
2 Définition de nom de types . . . 21

IX Modularité et portée des variables **21**
1 Modularité . . . 21
2 Organisation d'un module . . . 22
3 Exemple de module . . . 22
4 Portée des variables . . . 23

X Utilitaires **24**
1 Mode d'emploi . . . 24
2 Entrées/sorties standard : stdio.h . . . 24
3 Traitement de chaînes : string.h . . . 26
4 Nombres aléatoires : stdlib.h . . . 27

XI Extras **27**
1 Unions . . . 27
2 Variables "statiques" rémanentes . . . 28
3 Mots-clés `const` et `volatile` . . . 28
4 Mot-clé `enum` . . . 29
5 Pointeurs de fonctions . . . 29

Avertissement

Ce document présente un résumé des caractéristiques essentielles du langage C, et ne constitue pas un manuel de référence exhaustif. Mes sources sont été essentiellement le manuel de référence historique de Ritchie et Kernighan, ainsi que l'excellent "Cours de programmation — Langage C" de Bernard Leguy, qui fût mon professeur à l'Université de Lille I. Les éventuelles erreurs et maladresses sont par contre de mon fait.

I Introduction

1 Origines

Le langage C a été conçu au début des années 1970 par Dennis Ritchie chez Bell Telephone Laboratories. L'objectif était de disposer d'un langage souple et efficace pour remplacer l'assembleur dans l'écriture de code pour le système d'exploitation Unix. De ces origines, le langage C a donc hérité les particularités suivantes :

- c'est un langage procédural, muni de structures de contrôle (*if*, *for*) ressemblant superficiellement à Pascal (conçu à la même époque), et aussi à Ada ;
- l'accès au données se fait à un niveau d'abstraction assez bas : par exemple passage d'adresses pour simuler les paramètres en entrées-sorties ;
- le typage et d'une manière générale le contrôle à la compilation est plutôt faible, avec beaucoup d'interprétations par défaut qui peuvent être à l'origine d'erreurs difficiles à corriger ;
- la syntaxe est parfois cryptique et permet des raccourcis scabreux ;
- c'est un langage efficace en ce sens que, en général, il ne masque pas la complexité des calculs : une expression simple conduit habituellement à un calcul simple.

Le premier livre de référence sur le langage C fut co-écrit par Ritchie et Kernighan, dont les initiales "K&R" servent à référencer cette version du langage. Par la suite la définition du C a été notamment complétée et améliorée en 1989 par l'ANSI [1], version reprise en 1990 par l'ISO [2] et connue sous les dénominations de C-ANSI,

1. American National Standards Institute.
2. International Organization for Standardization.

C89 ou C90. C'est cette version que nous utiliserons. L'ISO a émis un nouveau standard, qui emprunte des éléments au C++, en 1999, mais ce C99 reste encore peu supporté par les compilateurs actuellement.

2 Différences principales avec Ada

En général le langage Ada a déjà été vu lors des années précédentes. Le C s'en distingue notamment par les points suivants :

- le contrôle des types est plus réduit : beaucoup de conversions de types sont automatiques, même si elles conduisent à une perte de précision parfois importante ;
- il n'existe pas de type ensemblistes (les opérations "bit à bit" (*bitwise*) peuvent evéntuellement y suppléer) ;
- il n'y a pas de type intervalle ;
- les indices de tableaux sont toujours des entiers démarrant à 0 ;
- les fichiers sont toujours considérés comme des suites potentiellement infinies de caractères ;
- il n'y a pas de passage de paramètres en mode "in out" ou "out" : on utilise des adresses (pointeurs) pour le simuler ;
- on ne peut pas déclarer une fonction dans une autre fonction ;
- il n'existe pas l'équivalent du "with" pour éclaircir certaines utilisations des pointeurs ;
- les "packages" peuvent être simulés mais plus grossièrement par des modules basés sur le découpage de l'application en fichier ;
- il n'y a pas de concepts "orientés objets"... mais le langage Ada initial (Ada83) n'en contenait pas non plus : si l'on veut des objets on peut utiliser C++ ;
- enfin, quand un programme Ada compile on est généralement proche de la solution, à moins d'une erreur de conception à la base. Mais quand un programme C compile, il peut rester de nombreuses bogues cachées par les interprétations par défaut du langage.

3 Quelques rappels de base

3.i Compilation

Le langage C est un langage compilé, c'est à dire que le programmeur édite un fichier texte décrivant son algorihtme en C, appelé fichier *source* d'extension ".c". Le compilateur analyse le source et crée un programme équivalent un langage machine, appelé fichier *objet*, d'extension ".o". Ce fichier objet devra encore être lié notamment avec du code d'initialisation dépendant du système et très souvent du code issu des bibliothèques standards qui complètent les possibilités du langage C (ex : entrées/sorties standards). Lorsque l'application que l'on écrit est assez grande, on la divise en plusieurs "modules" qu'il faut aussi lier entre eux. Après cette "édition de liens", on obtient finalement un fichier "exécutable".

Note : en TP nous utiliserons le compilateur gcc (Gnu C Compiler) qui est un logiciel libre. Dans le cas où l'on ne compile qu'un seul fichier, l'édition de lien sera intégrée dans l'appel au compilateur et le fichier exécutable sera généré en une fois :

```
gcc -o nom_executable source.c -Wall
```

Note : l'option -o signifie "output" (et pas "objet" !) et sert à préciser le nom de l'exécutable en sortie (sinon il porte le nom par défaut "a.out"). L'option -Wall (pour "warnings all") demande au compilateur d'afficher tous les avertissements relatifs à la syntaxe et à la sémantique du code compilé : il s'agit de tournures présentes dans votre source qui sont légales dans le langage mais qui toutefois peuvent recéler une erreur de programmation. Il est extrêmement recommandé d'afficher tous les avertissements quand on programme en langage C (malgré tout des erreurs peuvent rester dissimulées...).

3.ii Gestion de la mémoire

Le C étant proche de la machine pour les données, il est utile d'avoir une bonne idée de l'organisation de la mémoire de la machine. On peut imaginer la mémoire d'un ordinateur comme une série de cellules capables de contenir chacune un octet (c.à.d. un nombre binaire de 8 symboles) et étiquetées chacune par une adresse, les adresses allant de 0 à la capacité mémoire de la machine -1 (la réalité de la gestion mémoire est plus compliquée mais cette abstraction nous suffira). Tout objet du langage (variable simple, tableau, fonction) est rangé en mémoire où il occupe un ou plusieurs octets consécutifs : il peut donc être repéré par l'adresse du premier octet mémoire qu'il occupe.

Enfin on divise de manière abstraite la mémoire utilisée par le programme en 3 parties :

- une partie "statique" déterminée à la compilation et qui contient le code des fonctions et les autres objets globaux du programme ; ces objets existent pendant toute la durée du programme ;
- une partie appelée "pile" qui sert à héberger temporairement les paramètres des fonctions et les variables locales lors d'un appel de fonction ; au moment de l'appel la pile s'accroît pour stocker ces objets, puis à la fin de l'exécution de la fonction la pile diminue et ces objets temporaires sont détruits ;

- enfin une partie dynamique appelée "tas" qui sert à allouer des objets à la demande du programme, par le biais d'appel aux fonctions de la famille "malloc" (allocation dynamique) que nous verrons plus loin ; à partir de leur création ces objets existent jusqu'à la terminaison du programme, sauf si l'on demande explicitement leur suppression (ou "libération").

II Eléments de base

1 Notions élémentaires

1.i fonction `main`, instructions, bloc, expressions élémentaires

- Tout programme contient une fonction de nom `main` qui indique le point de départ de l'exécution du programme.
- Partout où l'on peut mettre une instruction, on peut placer un bloc (une séquence) d'instructions qui sera mis entre accolades `{ }`. Le code d'une fonction (comme `main`) est toujours placé dans un bloc.
- Les instructions se terminent par un point virgule `;`
- L'affectation (ranger une valeur dans une variable) se note `variable = expression;`
- Les 4 opérations arithmétiques se notent `+ - * /` tandis que le modulo se note `%` et n'accepte que des opérandes entiers. La division est entière si et seulement si ses 2 opérandes sont entiers.

1.ii Commentaires

On ouvre un commentaire avec la séquence `/*` et on le ferme avec `*/`. Un commentaire peut contenir toute suite de caractères (sauf `*/`), y compris des sauts de lignes. Un commentaire est considéré comme un espace par le compilateur, et donc on peut même les utiliser pour séparer des instructions dans une même ligne, ce qui est évidemment déconseillé pour des raisons de lisibilité. Exemple de commentaire :

```
/* ceci est un commentaire C
sous forme
    traditionnelle */
```

Par ailleurs, bien que cela ne fasse pas partie du standard C89, les compilateurs récents (notamment `gcc`) acceptent souvent aussi la même notation que le language C++ pour les commentaires. Un tel commentaire débute par `//` et se continue jusque la fin de ligne . Exemple :

```
// ceci est un commentaire sous forme C++
// cecie est un autre commentaire
```

2 Variables, littéraux et types simples

Typage et nommage. Les variables servent à stocker les données manipulées par votre programme. Bien que C permette d'utiliser des variables sans les déclarer à l'avance (en considérant alors qu'elles sont de type entier par défaut), c'est une très mauvaise pratique : on déclarera toujours les variables en leur donnant donc un type explicitement. Les déclarations de variables sont des instructions et se terminent donc par un point-virgule `;` . Elles sont toujours de la forme : `type_variable nom_variable;`

- Les noms de variables (et autres : types, fonctions...) sont de la forme : lettre ou tiret bas `_` suivi d'un nombre quelconque de lettres, tirets bas ou chiffres. Cependant les noms commençant par un tiret bas sont à éviter car en principe réservés aux librairies livrées avec le compilateur.
- Attention, C fait la différence entre majuscules et minuscules : `toto` et `Toto` sont 2 noms différents. Les mots-clés du langage sont toujours en minuscules, les noms de types ou de constantes de certaines bibliothèques sont parfois en majuscules.
- Convention : on utilisera les minuscules pour les noms de variables, sauf en cas de nom composé de plusieurs mots (par ex : `notePrincipale` ou `compteurDeBoucles`). Les majuscules seront réservées aux constantes (ex : `PI`, `TAILLE_MAX`) ;
- Une variable peut être initialisée lors de sa déclaration : `type_variable nom_variable = valeur_initiale;`
- Plusieurs variables de même types peuvent être déclarées à la suite, avec d'éventuelles initialisations : `type_variable nom_variable_1, nom_variable2 = valeur, nom_variable_3;`

Emplacement des déclarations. Les déclarations de variables se font :

- soit en dehors de tout bloc et fonction, au niveau du fichier ; les variables déclarées ainsi sont alors par défaut globales à tout le programme (même s'il se compose de plusieurs fichiers), et existent pour toute la durée d'exécution.

- soit au début d'un bloc (c'est notamment le cas de toutes les variables déclarées dans une fonction). Les variables sont alors locales au bloc, et sont dites "automatiques : elles sont créées automatiquement à l'entrée du bloc et détruites lorsque l'on en sort.
  Note : la règle de déclaration en début de bloc a été assouplie en C99, et l'on peut désormais déclarer les variables locales partout où on peut placer une instruction (ces variables restent locales et automatiques).
- on peut déclarer deux ou plus variables de même nom, à condition qu'elles ne soient pas déclarées au même niveau d'imbrication (en terme de bloc). Une telle variable déclarée dans un bloc imbriquée prend le pas, à l'intérieur du bloc, sur les variables déclarées au niveau supérieur. Les variables de niveau supérieur sont simplement *masquées*, elles continuent d'exister et sont à nouveau accessible à la fin du bloc qui les masque (voir exemple ci-dessous).

```
int a; // a est une variable qui peut contenir des entiers (int)
int b;

int main() { // début du programme principal

  a= 12; // range 12 dans a

  { // ouverture d'un bloc imbriqué

    int a; // création d'une nouvelle variable a

    a = 5;

  } // fermeture du bloc imbriqué : le a du bloc est détruit

  b = a; // b reçoit 12, contenu dans la variable a
}
```

2.i Les caractères

Déclaration de 3 variables de type caractère et de nom k, k0, et k1 initialisée avec le caractère 'a' :
`char k, k0, k1 = 'a';`

Les littéraux de type caractères (ou constantes littérales, c.à.d. constantes dont la valeur est écrite "en toutes lettres") se notent entre apostrophes, par exemple : `'a'  'X'  '1'  ';'`

Certains littéraux correspondent à des caractères non imprimables ou qui sont interprétés par votre éditeur de texte, par exemple le retour chariot, ou par le compilateur. On doit donc leur donner une représentation spéciale pour pouvoir les manipuler en tant que simples données caractères : ils sont codés sur 2 caractères dont le premier est l'anti-slash (mais ne reprèsentent qu'un seul caractère). Exemple :

siginification	saut de ligne	tabulation	effacement arrière	retour chariot	saut de page	anti-slash	apostrophe
codage en C	`'\n'`	`'\t'`	`'\b'`	`'\r'`	`'\f'`	`'\\'`	`'\''`
valeur 'ASCII'	LF	TAB	BSP	CR	FF	\	'
code octal	12	11	10	15	14	134	47
code décimal	10	9	8	13	12	92	39

On note aussi les littéraux caractères grâce à leur code ASCII sur 3 chiffres en octal (base 8) précédé d'un anti-slash : `'\nnn'`

Enfin le caractère anti-slash \ est ignoré s'il est placé devant un caractère ordinaire : `'\w'` <=> `'w'`

2.ii Les entiers

Déclaration de 3 variables entières, dont 2 sont initialisées : `int i, j=5, w=0;`

Par défaut la base utilisée pour les littéraux entiers est le décimal, mais attention, un littéral entier qui commence par 0 est interprété comme étant écrit en octal (base 8, et donc ne devrait contenir que des nombres entre 0 et 7) !

Il est possible de noter les littéraux en hexadécimal (base 16) en les préfixant par `0x` ou `0X` suivi de chiffres dans l'ensemble 0 à 9 ainsi que A à F en majuscules ou minuscules pour représenter les chiffres hexadécimaux entre 10 et 15 décimal. Ainsi par exemple :

`64` (décimal) = `0100` (octal) = `0X40` (hexa)
`63` (décimal) = `077` (octal) = `0x3f` (hexa)

Généralement les ordinateurs actuels - 2009 - utilisent des `int` codés sur 32 bits, qui peuvent représenter les valeurs dans l'intervalle $[-2^{31}, 2^{31} - 1]$. Toutefois le C prévoit différents types d'entiers de capacité plus ou moins grande, mais leur implantation dépend du compilateur et de la machine utilisée ! On garantit seulement que les valeurs d'un type "plus petit" sont incluses au sens large dans un type "plus grand" :

`char` <= `short` <= `int` <= `long`

Notez que le type `char` est considéré comme un type entier : en effet les caractères sont stockés en mémoire par leur code numérique (dans un des standards ASCII, ISO, UTF... selon le système utilisé) et donc C les considère comme tels. Ainsi on peut écrire par exemple : `char c = 64;` ce qui revient à écrire `'A'` dans la variable `c`. Par la suite l'affectation `c = c + 1;` remplace le caractère actuel par le suivant dans le code (ici `'B'` dont le code est 65).

Enfin les types entiers peuvent être préfixés par le mot-clé `unsigned` : ils sont alors considérés comme non signés, c'est à dire représentant uniquement des positifs dans l'intervalle $[0, 2^{taille} - 1]$, où taille est la capacité en bits du type considéré sur la machine considérée.

2.iii Les réels

Ils sont de type :

- `float` : simple précision ;
- `double` : double précision ;
- `long double` : quadruple précision.

Les littéraux réels s'écrivent avec 1 point décimal et au moins un chiffre à droite ou à gauche. La notation scientifique avec exposant de 10 est acceptée auquel cas le point décimal n'est plus obligatoire.

Un littéral réel est :

- de type `double` par défaut ;
- de type `float` s'il est suivi de `f` ou `F` ;
- de type `long double` s'il est suivi de `l` ou `L` ;

Exemples :

écriture	**interprétation**
15.75F	un float
15.75	un double
15.75L	un long double
1.575E1	= 15.75
1575E-2	= 15.75
-2.5e-3	= -0.0025
25e+4	= 250000

2.iv Les booléens

Il n'existe pas de type booléen {vrai, faux} défini en C, quoiqu'il soit possible d'inclure une bibliothèque adéquate à partir du ISO C99. Nous nous en tiendrons au standard et utiliserons simplement des variables entières pour représenter les booléens : dans ce cadre la valeur 0 vaux faux et toute valeur différente de 0 vaut vrai.

2.v Les littéraux chaînes

Un littéral chaîne s'écrit entre guillemets :

`"un exemple de chaîne"`.

Tous les littéraux caractères sont autorisés, et on utilise `\"` pour insérer un guillemet dans la chaîne.

Ex : `"cette chaîne se finit par un saut de ligne\n"`.

Attention à ne pas confondre le caractère `'x'` et la chaîne `"x"` qui sont de types incompatibles et représentés en mémoire de façon différente.

2.vi Le pré-processeur

Avant la compilation à proprement parler le fichier source passe par un "pré-processeur" qui réalise essentiellement des opérations similaires à du copié-collé. C'est lui insère les entêtes des bibliothèques utilisées, notamment celle des entrées/sorties standards **stdio.h** comme on le verra ci-après. Il permet aussi de définir des constantes, de faire de la compilation conditionnelle (certaines instructions seront incluses selon la machine, le compilateur, etc). C'est lui aussi qui supprime les commentaires avant la compilation.

Pour le pré-processeur on ne parle pas d'instructions mais de *directives*. Elles sont repérées par leur 1er caractère qui est toujours `#`. Exemple :

```
// inclusion de la librairie d'E/S standard
#include <stdio.h>
```

Le pré-processeur permet aussi de définir des constantes de la façon suivante :

```
#define PI 3.141592
```

Le premier mot suivant `#define` est le nom de la constante, tout ce qui suit sur la ligne jusque la fin de ligne constitue sa valeur. Lors de la compilation, le pré-processeur fera un chercher-remplacer et transformera ici toute apparition de l'identificateur `PI` en `3.141592`. Attention, comme il ne s'agit pas d'une instruction à proprement parler, mais d'une directive du pré-processeur, pas de point virgule en fin de ligne.

2.vii Entrées/sorties au terminal

Les opérations d'entrée/sorties (E/S) ne font pas partie du langage, il faut placer en tête du fichier source la directive d'inclusion de la bibiothèque standard d'E/S :

`#include <stdio.h>`

On accède alors notamment aux fontions `scanf` et `printf` qui réalisent respectivement des saisies au clavier et l'affichage sur la console. En paramètre de ces fonctions on trouve d'abord une chaîne contenant des *formats* comme `%d` ou `%s` qui signifie respectivement qu'on va travailler avec un entier et avec une chaîne (string). Viennent ensuite les variables ou les expressions qui concordent avec ces formats, dans l'ordre de leur apparition, comme illustré dans l'exemple de programme ci-dessous, en section 3.

3 Exemple de programme source

```
#include <stdio.h>

// définition de constante
#define MAX 100

int a; /* var globale, type entier */

/* fonction principale, qui va retourner un entier */
/* tout programme contient une fonction principale */
/* c'est là que débute l'exécution du programme */
int main() {
  int b, c; /* 2 vars locales à la fonction main */
  char nom[MAX]; /* tableau de MAX caractères */

  printf("Quel est votre nom ?\n"); /* sortie d'une chaîne */
                                    /* avec saut de ligne */
  scanf("%s", nom); /* saisie d'une chaîne dans le tableau nom */
                    /* Note: pas besoin de & avant "nom" ici */
                    /* car c'est un tableau */


  printf("Bonjour %s\n", nom);  /* le format %s est remplacé par */
                                /* la chaîne contenue dans nom */


  printf("Entrez 2 entiers : ");
  scanf(" %d %d", &a, &b); /* le format %d veut dire entier décimal */
                           /* attention à ne pas oublier les & */

  c = a + b; /* c contient la somme de a et b */
  printf("la somme est : %d\n", c);

  return 0; /* sous Unix on retourne un entier à la fin 0 = ok */
} /* fin de main */
```

III Contrôle du flot d'instructions

Par défaut les instructions sont exécutées en *séquence*, c'est à dire l'une après l'autre dans l'ordre (occidental) de lecture : de gauche à droite et de haut en bas. Pour qu'un langage soit généraliste (théorème de Böhm-Jacopini) il faut qu'il comprenne la séquence, l'alternative (le "si") et, soit la boucle, soit la récursivité. Ce sont ces structures de *contrôle du flot d'instructions* (qui influent sur l'ordre d'exécution des instructions) que nous voyons dans la suite.

A noter que la mise en forme du texte source importe peu : plusieurs instructions peuvent être à la suite sur une même ligne, et là où on peut mettre un espace on peut passer une ou plusieurs lignes. Néanmoins, pour préserver la lisibilité, on écrit généralement une seule instruction par ligne.

1 Expression du choix : alternative et aiguillage

1.i Alternative — if

La construction "if" se note :

```
if (condition)
  instruction-ou-bloc
```

L'alternative peut contenir une partie "sinon" :

```
if (condition)
  instruction-ou-bloc
else
  instruction-ou-bloc
```

Exemple :

```
if (a > b)
  printf("a est plus grand que b.\n");
else { /* ouvrir un bloc si plusieurs instructions */
  printf("a  est plus petit");
  printf(" ou égal à b.\n);
}
```

1.ii Aiguillage — switch

Il est fréquent (par exemple : gestion d'un menu) de devoir exécuter une tâche selon la valeur d'une expression discrète. On utilise soit une série de `if`, soit une structure d'aiguillage notée `switch` :

- Notez qu'on peut avoir dans un **case** une séquence d'instructions sans qu'elle soit dans un bloc `{ }` (c'est une exception à la règle habituelle).
- L'oubli du mot-clé "break" indique à C de poursuivre par l'exécution du cas suivant, ce qui en général n'est pas souhaité, sauf lorsque plusieurs cas ont le même traitement.
- "default" récupère toutes les valeurs non testée plus haut. Il est fortement recommandé de toujours spécifier un traitement par défaut.

Exemple :

```
...
/* on souhaite tester la variable "a" de type entier */
switch (a) {
  case 0 : /* on groupe plusieurs cas, en ommettant "break" */
  case 1 :
```

```
switch (expression-discréte) {
  case valeur_1 : séquence-d-instructions break;
  case valeur_2 : séquence-d-instructions break;
  ...
  case valeur_n : séquence-d-instructions break;
default :
  séquence-d-instructions
}
```

```
  case 2 : printf("a <= 2\n"); break; /* on quitte l'aiguillage */

  case 3 : printf("a == 3\n"); break; /* on quitte l'aiguillage */

  default : printf("a >= 4\n");
} /* fin du switch */
```

2 Boucles

Comme indiqué plus haut, les boucles — ou la récursivité — sont nécessaires pour qu'un langage soit généraliste.

2.i Tant que — while

C'est la boucle la plus utilisée dès qu'on ne fait pas une simple énumération. Elle s'exprime sous la forme :

`while (` *condition-itération* `)`
  instruction-ou-bloc

Exemple :

```
/* affiche les entiers de 0 à 9 inclus */
int cpt;
cpt = 0;
while (cpt < 10) { // bloc car plus d'une instruction
  printf("%d\n", cpt);
  cpt = cpt + 1;
}
```

Note : si la condition d'itération est fausse au début de la boucle, aucune itération n'est effectuée.

2.ii Pour — for

La boucle "for" devrait en principe être réservé à des cas de simple énumération d'un type discret. Typiquement, si on ne connait pas à l'avance le nombre d'itérations, ou si le compteur de boucle ne parcourt pas une suite simple de valeurs, alors il sans doute préférable d'utiliser la boucle "while" traitée précédemment.

`for (`*initialisation* `;` *condition-continuation* `;` *variation-compteur* `)`
 instruction-ou-bloc

Exemple :

```
/* affiche les entiers de 0 à 9 inclus */
int cpt;
for (cpt = 0; cpt < 10; cpt = cpt + 1) /* pas de bloc car une seule instruction */
  printf("%d\n", cpt);
```

Notes : Le code de cet exemple a exactement la même signification que l'exemple du **while** vu plus haut. Si la condition de continuation d'itération est fausse dès le début de la boucle, aucune itération n'est effectuée.

2.iii Faire tant que — do while

La boucle "do while" est probablement une exception dans les langages de programmation : habituellement il s'agit plutôt d'un "répéter jusqu'à" où l'on donne la condition d'arrêt, alors qu'en C c'est la condition de continuation ! Par ailleurs, il est souvent sage dans un programme de tester s'il est possible d'itérer avant de commencer à le faire, c'est pourquoi le "do while" est la structure de boucle la moins utilisée. Avant d'écrire un "do while" posez vous la question : faut-il *toujours* exécuter au moins une fois les intructions de la boucle ?

`do`
 instruction-ou-bloc
`while (` *condition-itération* `);`

Exemple :

```
/* affiche les entiers de 9 à 0 inclus */
int cpt;
cpt = 9;
do { /* bloc car plus d'une instruction */
  printf("%d\n", cpt);
  cpt = cpt - 1;
} while (cpt >= 0);
```

- Attention à ne pas oublier le point-virgule en fin de construction.
- Notez que si par erreur la variable "cpt" avait été initialisé à -1, nous l'aurions affiché, car le test de validité est réalisé en fin de boucle, après l'affichage... À bon entendeur...

2.iv Compléments sur les boucles

Deux instructions restent à découvrir relativement aux boucles, le `break;` et le `continue;`.

- `break;` a déjà été vu relativement à l'instruction `switch`, mais il peut aussi s'utiliser dans une boucle, où il donne l'ordre de quitter la boucle *immédiatement* englobante (c'est à dire que s'il y a plusieurs boucles imbriquées l'une dans l'autre, on ne sort que de la boucle la plus intérieure).
- `continue;` modifie aussi le déroulement de la boucle en faisant se poursuivre l'exécution immédiatement au début de boucle : on passe en fait immédiatement à l'itération suivante.

Exemple :

```
/* affiche les entiers de 9 à 0 inclus */
int cpt;
cpt = 12;
do { /* bloc car plus d'une instruction */
  cpt = cpt - 1;
  if (cpt > 9) /* inutile d'afficher */
    continue;
  if (cpt < 0) /* quitter */
    break;
  printf("%d\n", cpt);
} while (1); /* itérer toujours (1 veut dire "vrai") : on sort avec le break */
```

3 Sauts et labels

Ces constructions, héritées de la programmation en langage d'assemblage, ne sont que très rarement justifiées (ex : écriture d'un analyseur lexical — encore que l'utilisation d'un logiciel de type "lex" soit bien préférable, traitement d'erreurs). Elle sont donc à éviter et ne sont mentionnées qu'à titre de référence.

- toute instruction peut être préfixée par une *étiquette* qui à la forme d'un identificateur : `étiquette : instruction`
- dans la fonction qui contient l'étiquette, on peut placer des instructions de *saut* pour dérouter le cours du programme qui se poursuivra à partir de l'instruction étiquetée : `goto étiquette;`

IV Expressions

1 Définition

Une expression se lit comme une valeur plutôt que comme une instruction. En tant que valeur, une expression est généralement utilisée dans un calcul, une comparaison, ou est affectée à une variable. Une expression peut être une simple constante littérale, un nom de variable ou de constante interprété alors comme son contenu, un appel de fonction remplacé par la valeur retournée par la fonction. Enfin ces élément peuvent être combinés par des opérateurs, que nous allons voir en détail.

Le langage C possède 15 niveaux de priorité d'opérateurs, ce qui rend l'usage de parenthèses utile en cas de doute, ou simplement pour améliorer la lisibilité du code. Certains opérateurs modifient leurs opérandes, ou bien sont habituellement des instructions dans les autres langages, ce qui les rend particulièrement délicat à utiliser.

2 Opérateurs (listés par priorité décroissante)

classe	opérateur
primaires	`[indice] (paramètres) . ->`
unaires	`* & (type) sizeof(var-ou-type) - ~ ! ++ --`
multiplicatifs	`* / %`
additifs	`+ -`
décalages	`<< >>`
ordres	`< <= > >=`
égalités	`== !=`
et binaire	`&`
ou exclusif binaire	`^`
ou binaire	`|`
et logique	`&&`
ou logique	`||`
conditionnelle	`?:`
affectations	`= += -= *= /= %= <<= >>= &= ^= |=`
séquence	*virgule* `,`

primaires :

`[indice]` opérateur d'indiçage (accès au ième élément d'un tableau) : `a[10] = a[10] * 2;`

`(paramètres)` passage de paramètres lors d'un appel de fonction : `a = abs(a);`

`.` accès à un champ d'un enregistrement : `e.valeur = 10;`

`->` accès à un champ d'enregistrement pointé : `pe->champ = 10; /* pe est un pointeur d'enregistrement */`

unaires :

`*` dépointage (passage d'une adresse à l'objet pointé) : `(*pe).champ` est équivalent à `pe->champ`

`&` adressage (obtient l'adresse d'une variable, inverse du précédent) : `*(&a)` est équivalent à `a`

`(type)` conversion dans le type spécifié : `(int) 3.14` vaut `3`

`sizeof(var-ou-type)` vaut la taille en octets de l'objet ou du type passé en paramètre : `sizeof(char)` vaut 1 en général

`-` le "moins" unaire comme dans : `-3`

`~` le complément binaire : `~(1001)` vaut `0110`

`!` négation logique : `!(a > b)` équivaut à `(a <= b)`

`++` incrémentation d'une variable entière ; attention deux usages possibles selon la position de l'opérateur :

- `b = a++` équivaut à `b = a; a = a + 1;`
- `b = ++a` équivaut à `a = a + 1; b = a;`

`--` décrémentation d'une variable entière ; attention deux usages possibles selon la position de l'opérateur :

- `b = a--` équivaut à `b = a; a = a - 1;`
- `b = --a` équivaut à `a = a - 1; b = a;`

multiplicatif : rien de spécial, voir aussi 1.i

additifs : rien de spécial

décalages : à droite ou à gauche, au niveau binaire, d'autant de bits que le 2ème opérande ; soit `A` une variable charactère (rappel : char fait partie des types entiers voir 2.i) contenant la valeur décimale 11 c'est à dire `00001011` en binaire :

- `A << 1` vaut `00010110` : les bits sont décalés à gauche et un `0` est inséré à droite (le bit le plus à gauche — bit de poids fort — est perdu). Note : équivalent à une multiplication par 2 si le bit le plus à gauche était 0.
- `A >> 1` vaut `00000101` : les bits sont décalés à droite (on perd le bit le plus à droite — bit de poids faible) et un `0` est inséré à gauche. Note : équivalent à une division entière par 2.

ordres : pour les comparaison de valeur numériques, rien de spécial

égalité :

`==` signifie "égal" dans les comparaisons (attention, ne s'applique pas aux tableaux ni aux chaînes de caractères)

`!=` signifie "différent", avec les mêmes restrictions

et binaire : `1001 & 0011` vaut `0001`

ou exclusif binaire : `1001 ^ 0011` vaut `1010`

ou binaire : `1001 | 0011` vaut `1011`

et logique : `&&` est employé dans les conditions

ou logique : `||` est employé dans les conditions, sa priorité est inférieure à celle de `&&`, conformément à l'interprétation dans l'algèbre de Boole

conditionnelle : il s'agit d'une sorte de "if" qui calcule un résultat, l'expression conditionnelle est avant le `?`, tandis que le `:` sépare la valeur dans le cas vrai de celle dans le cas faux : `(a < b) ? a : b` à pour valeur le minimum de `a` et `b` ;

affectations :
- l'affectation ordinaire est une expression qui a pour valeur ce qui a été affecté, donc `a = 3` vaut `3` ; Par conséquent on peut chaîner des affectations : `a = b = c = 3;` les 3 variables contiennent `3` (notez que l'affectation est analysée de *droite à gauche*, c'est `c = 3` qui est effectué en premier).
- les autres versions cumulent l'affectation et une opération parmi celles déjà vues plus haut ; Ainsi `a += 3` équivaut à `a = a + 3`, de même `a -= 3` équivaut à `a = a - 3`, etc.

séquence : on peut toujours effectuer plusieurs expressions à la suite en les séparant par une virgule, le tout reste une expression ayant pour valeur la dernière sous-expression ;

Exemple : `d = (a = 5, scanf("%d",&b), c = 12);`

cette expression range 5 dans `a`, effectue la saisie de `b`, range 12 dans `c` et finalement range aussi 12 dans `d` (noter l'appel à la fonction `scanf()` qui est une expression) ; à éviter pour conserver la lisibilité !

V Pointeurs, tableaux statiques à 1 dimension, chaînes de caractères

1 Pointeurs

Un pointeur (plus précisément une variable pointeur) est une variable qui peut contenir l'adresse mémoire d'un "objet", par exemple une variable ou une fonction. Par exemple un pointeur peut contenir l'endroit dans la mémoire (l'adresse mémoire) où une variable entière range la donnée qu'elle contient. En utilisant cette adresse on peut directement modifier la valeur rangée dans la variable, sans utiliser le nom de la variable.

En C les adresses sont typées, c'est à dire qu'on doit préciser quel est le type d'objet dont on manipule l'adresse (on parle du type de l'objet pointé). La déclaration d'un pointeur utilise l'étoile * (même symbole que la multiplication), et est de la forme :

`type_pointé * variable_pointeur;`

Accéder à l'objet pointé se dit "dépointer", et utilise aussi l'étoile placée devant la variable pointeur, comme indiqué dans les exemples qui suivent.

```
int main() {
int a, b, *p; // a et b sont des entiers, p est un pointeur d'entiers
int *q, c; // q est un pointeur d'entier, c est un entier
int **pp ; // pp est un pointeur de pointeur d'entier
p = &a; // p contient l'adresse de a
*p = 2; // équivaut à: a = 2;
*q = 2; // ERREUR : q n'est pas initialisé, il pointe n'importe où
q = p; // q pointe aussi vers a
c = *q; // équivaut à: c = a;
p = &q; // ERREUR : p est un pointeur sur entier et pas sur pointeur d'entier
pp = &q; // OK
**pp = 3; // a vaut maintenant 3
*pp = &c; // q pointe sur c
&q = pp; // ERREUR : &q est l'adresse de q, c'est une valeur constante, pas une variable!
pp = &&a; // ERREUR : (&a) est une valeur et n'a pas d'adresse
q = &b; // q pointe sur b
*q = *p; // équivaut à: b = c;
}
```

2 Tableaux statiques à 1 dimension

Présentation. Un tableau est une collection ordonnée d'éléments (de variables) de même type rassemblés physiquement à la suite les uns des autres en mémoire, et auxquels on accède en donnant le nom de tableau et le numéro (on dit l'indice) de l'élément dans la collection. Un tableau est dit "statique" si il est complètement spécifié dans le code au moment de la compilation. Les tableaux dont la taille change pendant l'exécution du

programme sont dit “dynamiques”. On se limite ici aux tableaux statiques à une seule dimension, analogue aux vecteurs manipulés en mathématiques. Ils se déclarent sous la forme :

`type_d_élément variable_tableau[taille];`

Par exemple, pour déclarer un tableau de 10 entiers : `int tab[10];`

Les éléments sont toujours numérotés on dit (indicés) à partir de 0. Pour un tableau de taille N, les indices des éléments vont donc de 0 à N-1. Attention il n'existe pas dans le langage d'opérations fonctionnant sur la totalité du tableau : on ne peut pas en une seule opération copier un tableau dans un autre, ni comparer 2 tableaux, ni demander à un tableau quelle est sa taille!!!

Exemples :

```
int main() {
int t1[10], t2[10];
t1[0] = 12; // range 12 dans la 1ère case de t1
t1[9] = 5; // range 5 dans la dernière case de t1
t2[1] = t1[0]; // copie la 1ère case de t1 dans la 2ème case de t2
t1[10] = 9; // ERREUR t1[10] n'existe pas ! t1 est indicé de 0 à 9
t1 = t2; // ERREUR on ne peut pas affecter le contenu de t2 dans t1
{
  int i;
  for (i=0; i < 10; i++)
    t1[i] = t2[i];

} // OK, copie de t2 dans t1
```

3 Lien entre pointeurs et tableaux.

On peut considérer un nom de tableau comme l'adresse en mémoire de son premier élément. Un nom de tableau peut donc être assimilé à une sorte de pointeur constant (qu'on ne peut pas modifier). On peut donc accéder à ses éléments par une opération de dépointage. Exemples :

```
int main() {
int t1[10], t2[10];
*t1 = 12; // range 12 dans la 1ère case (car t1 est l'adresse de cette case)
*(t1 + 9) = 5; // range 5 dans la dernière case de t1 (9 cases après la première)
*(t2 + 1) = *t1; // copie la 1ère case de t1 dans la 2ème case de t2
*(t1 + 10) = 9; // ERREUR t1[10] n'existe pas !
{
  int i;
  for (i=0; i < 10; i++)
    *(t1+i) = *(t2+i);

} // copie de t2 dans t1
```

Le code de copie d'un tableau dans un autre, donné juste au dessus en exemple peut aussi être exprimé en copiant les adresses des tableaux dans des variables pointeurs :

```
int main() {
  int t1[10], t2[10];
// on suppose que t2 est rempli par des données, à copier dans t1
  int i, *p1=t1, *p2=t2;
  for (i=0; i < 10; i++)
    *p1++ = *p2++;
}
```

A noter que l'on ne gagne pas vraiment en lisibilité! C'est possible mais à éviter...

4 Variables chaînes de caractères

Les chaînes de caractères, que je noterai simplement “chaînes”, sont un type de données très utilisé, notamment pour l'affichage et la saisie de messages, noms, bref pour les données textuelles.

En C une variable chaîne n'est rien d'autre qu'un tableau de caractères à 1 dimension, et qui contient un caractère spécial marquant la fin de la chaîne. Si l'on veut stocker un message de 20 caractères, on doit donc utiliser un tableau d'au moins 21 caractères (donc 1 caractère de plus), pour réserver la place de ce marqueur de

fin de chaîne. Le tableau peut sans problème être plus grand : grâce au marqueur de fin, s'il y a des caractères superflus en fin de tableau, situés derrière le marqueur de fin, ils seront ignorés par les fonctions de manipulation et d'affichage de chaîne.

Le marqueur de fin est par convention le caractère (non affichable ni imprimable) de code 0. On peut le noter `'\0'`, mais comme le type caractère est un vu par C comme un type numérique, on peut aussi l'écrire 0, mais attention en aucun cas '0' ni "0"!!! ATTENTION, sans ce marqueur, un tableau n'est pas une chaîne et ne peut pas être manipulé par les fonctions standards sur les chaînes !

Toutes les possibilités et restrictions précédemment mentionnées pour les tableaux sont aussi valables pour les chaînes : pas de copie ni de comparaison, utilisation possible de pointeurs.

```
int main() {
char ch1[10]="012345678"; // attention à bien respecter le nombre d'éléments!
// il faut penser au marqueur de fin de chaîne qui utilise 1 caractère!!!
char ch2[10];

ch2[0] = ch1[0]; // ch2[0] vaut maintenant '0'
*(ch2 + 1) = 'B'; // ch2[1] vaut 'B'
printf("%s\n", ch1); // OK
printf("%s\n", ch2); // ERREUR on n'a pas mis de marqueur de fin dans ch2
ch2[2] = 0; // ch2 est maintenant une vraie chaîne
printf("%s\n", ch2); // OK, affiche "0B"
}
```

La présence du marqueur de fin permet de connaître la longueur de la chaîne (qui peut être plus petite que celle du tableau qui la contient). Exemple de copie de chaine :

```
int main() { // copie raisonnable
  char ch1[10]="012345678";
  char ch2[10];
  int i = 0;
  while (ch1[i] !=0)
    ch2[i] = ch1[i++]; // on incrémente i pour l'itération suivante
  ch2[i] = 0; // on place la fin de chaîne dans ch2
}
```

Comme 0 est aussi interprété comme le booléen FAUX dans les tests, on peut encore compacter l'écriture en utilisant des pointeurs :

```
int main() { // copie raccourcie
  char ch1[10]="012345678";
  char ch2[10];
  char *p1 = ch1, *p2 = ch2;
  while (*p1) // donc tant que (*p1 != 0)
    *p2++ = *p1++; // on incrémente pour l'itération suivante
  *p2 = 0; // on place la fin de chaîne dans ch2
}
```

Et comme l'affectation est vue comme une expression renvoyant la valeur affectée, on peut même condenser en :

```
int main() { // copie pour "geek"
  char ch1[10]="012345678";
  char ch2[10];
  char *p1 = ch1, *p2 = ch2;
  while (*p2++ = *p1++) // donc tant que (*p1 != 0)
    ; // il n'y a plus rien à faire ici

  // la copie du marqueur a été faite dans la boucle
}
```

Evidemment, là encore ce qui est faisable n'est pas pour autant recommandé : la lisibilité est le critère à privilégier pour 95% du code.

VI Fonctions

Quelques éléments essentiels des fonctions en C :

- il a peu de différence syntaxique entre fonction et "procédure" (à la ADA) : une procédure est simplement une fonction qui renvoie `void` (c'est à dire rien, néant) ;
- un appel ou une définition de fonction est **toujours** suivie d'une paire de parenthèses. Ces parenthèses entourent les paramètres, ou rien du tout s'il n'y a pas de paramètres ;
- les paramètre effectifs (lors de l'appel) sont toujours passés par valeur : pas d'équivalent du mode "out" de ADA ;
- pas de définition de fonction à l'intérieur d'une autre définition de fonction.

1 Syntaxe

Forme d'une définition de fonction :

```
type_retourné nom_de_fonction ( type_param_1 nom_param_1,
                                type_param_2 nom_param_2,
                                ...
                               )
{
  déclarations_de_variables_locales
  liste_d_instructions
}
```

- si la fonction ne doit rien retourner, le type retourné est `void`
- si le type retourné n'est pas `void`, alors il doit y avoir une instruction `return expression;` retournant le résultat ;
- si le type retourné est `void`, on peut trouver l'intruction `return;` sans expression associée, qui sert uniquement à quitter immédiatement la fonction (pas le programme).

Forme d'un appel de fonction :

```
nom_de_fonction ( param_effectif_1 , param_effectif_2, ... )
```

- tout paramètre entier plus court que `int` est traduit en `int`
- tout paramètre `float` est traduit en `double`

Déclaration préliminaire Une difficulté syntaxique se présente lorsque le programmeur veut appeler une fonction qui se trouve être déclarée **plus bas** dans le code que l'endroit de l'appel : n'ayant pas encore rencontrée la définition de la fonction que l'on appelle, le compilateur va générer un message d'erreur et le programme ne sera pas complètement compilé. Or on ne peut pas toujours placer les définitions de fonctions avant les appels, notamment dans le cas de deux fonctions s'appelant récursivement l'une l'autre.

Dans cette situation on va placer une "déclaration préliminaire" de la fonction en tête du programme avant l'appel. Cette déclaration consiste simplement à dupliquer l'entête de fonction sans son corps entre accolades, suivi d'un point-virgule :

```
type_retourné nom_de_fonction ( type_param_1 nom_param_1,
                                type_param_2 nom_param_2,
                                ...
                               ) ;
```

Le compilateur passe alors d'abord sur la déclaration préliminaire, lorsqu'il rencontre l'appel de la fonction il peut vérifier que le type et le nombre des paramètres correspondent bien, et il effectuera à nouveau cette vérification lorsqu'il rencontrera la définition complète de la fonction. Exemple :

```
// déclarations préliminaires en tête du code
int fonction_un(int n);
int fonction_deux(int n);

// défintions et appel des fonctions
int fonction_un(int n) {
  if (n > 0) {
```

```
    printf ("Dans fonction_un\n");
    fonction_deux(n-1);
  }
}

int fonction_deux(int n) {
  if (n > 0) {
    printf ("Dans fonction_deux\n");
    fonction_un(n-1);
  }
}

int main() {
 fonction_un(5);
 return 0;
}
```

Ces déclarations préliminaires sont très souvent utilisées, même lorsqu'elles ne sont pas strictement indispensables, car elles permettent de rendre le code compilable indépendamment de la position respective des appels et des définitions de fonctions. Elles sont de plus nécessaires pour découper une application en plusieurs fichiers (voir la section sur la modularité).

2 Quelques exemples de fonctions

Calcul de factorielle :

```
int factorielle(int n) {
  int i, resultat=1;
  for (i=2; i <= n; i++)
    resultat *= i;
  return resultat;
}
```

Calcul de factorielle récursif : proposé pour l'exemple, car la récursivité est sans intérêt pour ce problème.

```
int factRec(int n) {
  if (n <= 1)
    return 1;
  else
    return n * factRec(n-1);
}
```

Valeur numérique d'un caractère chiffre entre '0' et '9' :

```
int valNum(char c) {
  if (c >= '0' && c <= '9')
    return c - '0'; /* les chiffres ont des codes successifs */
  else /* erreur */
    return -1; /* par convention */
}
```

Procédure pour passer une ligne sur la console :

```
void passeLigne() {
  printf("\n");
}
```

3 Simuler les paramètres en mode "in out"

Passage par copie. Les paramètres sont toujours passé par copie. Ainsi dans le programme suivant, la valeur affichée sera 3 et pas 27 :

```
#include <stdio.h>
void valAuCube(int n) {
  n = n * n * n;  /* si n valait 3 alors il vaut 27 */
}

int main() {
  int a = 3;
  valAuCube(a);
  printf("%d", a);
  return 0;
}
```

En effet, c'est le paramètre formel de nom `n` (qu'on peut considérer comme une variable locale à la fonction `valAuCube`) qui a été modifié. La variable `a` du programme principal est restée inchangée.

Passage par adresse. Par contre, si l'on passe *l'adresse* de `a`, alors on va pouvoir changer sa valeur depuis une fonction :

```
#include <stdio.h>
void valAuCubeBis(int *n) /* n est un pointeur sur une variable de type int */
{
  /* si à l'adresse contenue dans n il y avait 3 alors il y a 27 */
  *n = (*n) * (*n) * (*n);
}


int main() {
  int a = 3;
  valAuCubeBis(&a); /* a est passé par adresse */
  printf("%d", a);
  return 0;
}
```

Syntaxiquement, la modification consiste à placer `&` dans l'appel devant le paramètre effectif, et à placer `*` devant chaque apparition du paramètre formel correspondant (en fait nous utilisons dans la fonction des variables pointeurs, pointant sur la variable du programme principal).

La question à se poser lorsqu'on écrit une fonction est la suivante : "les paramètres seront ils des variables que l'on souhaite modifier dans la fonction ?". Si la réponse est non, on utilise un passage par valeur, si c'est oui on utilise un passage par adresse, appelé encore passage par pointeur (si on voulait être plus précis on dirait que l'on passe *par copie l'adresse* de la variable). Notez que vous avez déjà utilisé ces appels par adresse avec la fonction de saisie `scanf()` qui évidemment doit modifier le contenu des variables passées en paramètres.

Cas particulier des tableaux Comme on l'a vu plus haut un nom de tableau peut être vu comme l'adresse de la zone mémoire où est stocké le contenu du tableau. Par conséquent passer un tableau en paramètre c'est passer son adresse, et tout passage de tableau est toujours un passage par adresse. Donc on en déduit que :

- on peut toujours modifier un tableau reçu en paramètre dans une fonction ;
- il est inutile de préfixer avec l'opérateur d'adressage `&` un nom de tableau que l'on vaut passer en mode "in out", puisque par défaut c'est déjà une adresse que l'on manipule. Les compilateurs un peu anciens signalent une erreur dans ce cas, certains plus récents sont plus compréhensifs.

Ainsi lorsque l'on veut saisir une chaîne pas besoin de `&`, car une chaîne est stockée dans un tableau :

```
int main() {
  char chaine[80];
  printf("saisir une chaîne :\n");
  scanf("%s", chaine); // sans &
  printf("%s", chaine);
}
```

4 Les macros du pré-processeur

Le pré-processeur permet de simuler des appels de fonctions que l'on nomme *macros* ou macro-instructions : en fait le corps de la fonction sera inséré dans le texte du programme à chaque appel (d'autres langages de

programmation ont une notion ressemblante : les fonctions *inline*). Le contrôle syntaxique de ces macros est minimal, **il faut donc les éviter** et en tous les cas les réserver à des usages très simples. Exemple :

```
#define CUBE(n) (n * n * n)

int main() {
  int a = 3;
  printf("%d", CUBE(a));
  return 0;
}
```

le texte du programme sera transformé par le pré-processeur en :

```
int main() {
  int a = 3;
  printf("%d", (a * a * a));
  return 0;
}
```

L'usager de macros doit conserver à l'esprit les points suivants :

- les pseudo-paramètres sont spécifiés entre parenthèses, séparés par des virgules *sans espaces* s'il y en a plusieurs, et sans type, juste après le nom de macro ;
- le corps de la macro est *tout* ce qui suit sur la ligne ; si la ligne devient trop longue, on peut placer un \ et continuer sur la ligne suivante ;
- comme on ne sait pas a priori dans quel contexte le corps de la macro va être recopiée, il est prudent de le mettre entre parenthèses, pour éviter les problèmes de priorité d'opérateurs ;
- l'appel est remplacé par une copie : on gagne du temps de calcul (un tout petit peu), mais le programme est plus long ;
- une macro n'étant pas une vraie fonction, elle n'aura pas d'adresse mémoire lors de l'exécution du code donc sera inutilisable avec un pointeur de fonction ; par ailleurs aucune récursivité n'est possible.

VII Tableaux dynamiques, tableaux à 2 dimensions

1 Tableaux dynamiques

Les tableaux vus précédemment avaient une taille fixée une fois pour toute lors de leur déclaration dans le code source du programme. C'est une méthode tout à fait légitime pour de nombreuses applications, mais qui trouve ses limites lorsqu'on ne connait pas la taille du tableau à manipuler au moment où l'on écrit le code (par exemple on fait du calcul de matrices mais que la taille des matrices ne sera connue qu'à l'exécution). Il existe donc un procédé pour créer des tableaux "dynamiquement" lors de l'exécution du programme (on parle d'allocation dynamique).

Pour celà on inclut la bibliothèque `stdlib.h` ou encore `alloc.h`. On ne déclare pas un tableau proprement dit, mais une variable pointeur, et on alloue (c.à.d. réserve) un bloc de mémoire de la taille nécessaire pour stocker le tableau. La fonction d'allocation standard est `malloc()`, qui prend en argument la taille du bloc mémoire à réserver, et retourne l'adresse du premier élément du bloc qu'il faudra conserver pour accéder au tableau. Pour connaître la taille du bloc mémoire à passer en paramètre à `malloc` il suffit de multipliser la taille du tableau (en nombre d'éléments) par la taille occupée par un élément en mémoire, laquelle s'obtient par le mot-clé `sizeof`. Enfin la fonction `malloc` retourne un pointeur de type générique `void*`, et il faut une conversion de type entre parenthèses pour éviter les messages d'avertissements du compilateur lorsqu'on sauve cette adresse. Exemple d'allocation dynamique d'un tableau d'entiers :

```
#include <stdlib.h>
int main() {
  int taille;
  int *tab; // c'est un pointeur d'entiers
  printf("Entrez la taille du tableau : ");
  scanf("%d", &taille); // on suppose que taille est > 0
  if (taille <= 0) // erreur
    return 1; // on quitte la fonction "main"
  // ici la taille est positive
  // allocation : on réserve "taille" entiers, noter la conversion de type
  // entre parenthèse avant "malloc"
  tab = (int*) malloc(taille * sizeof(int));
```

```
  // on peut maintenant utiliser tab comme un tableau
  tab [0] = 12;

  // après utilisation, on libère la mémoire
  free(tab);
  return 0;
}
```

Comme montré dans l'exemple, on peut rendre la mémoire au système d'exploitation ("désallouer") lorsque l'utilisation du tableau est terminée, avec la focntion `free()` qui prend en paramètre le pointeur qui a été utilisé comme tableau.

2 Tableaux à 2 dimensions

Nous nous limiterons pour les tableaux multidimmensionnels à ceux à 2 dimensions, sachant que les principes de base s'étendent naturellement. La déclaration d'un tableau statique bidimensionnel est de la forme :

`type_d_élément variable_tableau[taille_dim_1][taille_dim_2];`

L'accès aux éléments se fait de manière très semblable aux tableaux unidimensionnels. Exemple :

`tab2D[0][0] = 12;` affecte 12 à la case d'indice (0,0).

Néanmoins la structure des tableaux 2D est assez spécifique : on a affaire en fait à un tableau de lignes, c'est à dire un tableau de tableaux unidimensionnels (ce qui fournit les 2 dimensions souhaitées). Soit un élément de tableau de type `T`, on a vu que le nom d'un tableau de tels éléments est vu comme une adresse de `T`, c'est à dire un `T*`. Si on prend maintenant un tableau de tels tableaux, il y a à nouveau passage à l'adresse et on obtient un objet de type `T**` (un pointeur de pointeur de `T`) : c'est le type que nous devrons gérer lors d'une allocation dynamique. Nous donnons ci-dessous un exemple d'allocation statique et un d'allocation dynamique de 2 tableaux d'entiers bidimensionnels.

```
#include <stdlib.h>
int main() {
  int t1[5][10]; // tableau 2D statique d'entiers

  // exemples d'utilisation
  t1[0][0] = 5;
  t1[4][9] = t1[0][0];

  int **t2; // pour un tableau dynamique 2D d'entiers
  // allouer le tableau de tableaux
  t2 = (int**) malloc(5 * sizeof(int *));
  // allouer chacun des tableaux lignes
  {
    int i;
    for (i = 0; i < 5; i++)
      t2[i] = (int *) malloc(10 * sizeof(int));
  }
  // on peut maintenant utiliser t2 comme on utiliserait t1
  t2[0][0] = 5;
  t2[4][9] = t2[0][0];

  // rendre la mémoire au système après utilisation
  {
    int i;
    for (i = 0; i < 5; i++)
      free(t2[i]);
  }
  free(t2);

}
```

VIII Structures enregistrements, définition de nom de types

1 Structures enregistrements

Nous avons vus précédemment la notion de tableau qui permet de rassembler dans une même variable une collection d'éléments de même type auxquels on accède par leur indice. Il peut aussi être utile de manipuler des collections d'objets de types différents, ce qui n'est pas possible dans un tableau ordinaire. On utilise alors une variable "structure", qui joue un rôle similaire à celui des enregistrements (`record`) de ADA. La variable structure à un nom, chacun des éléments qu'elle contient est appelé "champs" et possède un nom de champs. On utilise la notation pointée pour accèder à un champ particulier d'une variable structure. Exemple :

```
struct client {
  char nom[25];  // 1er champ
  int numero;    // 2ème champ
  double compte; // 3ème champ
};

struct client cl1, cl2;    // désormais "struct client" est un type connu

// on peut déclarer des variables à la fin de la définition de la structure :
struct bidule{
  int machin;
  float truc;
} b1, b2; // b1 et b2 sont de type struct bidule

int main() {
  printf("entrez le nom du 1er client ");
  scanf("%s", cl1.nom);
  printf("entrez le numéro du 1er client ");
  scanf("%d", &cl1.numero);
  printf("entrez la valeur du compte du 1er client ");
  scanf("%f", &cl1.compte);

  // initialiser à vide le 2ème client
  cl2.nom[0] = 0; // chaîne vide
  cl2.numero = 0;
  cl2.compte = 0.0;

  // soustraire 10 euros au compte du client 1
  cl1.compte = cl1.compte - 10.0;

  //

  return 0;
}
```

Note : les noms de structure et de champ sont gérés dans un espace de noms différents de celui des autres identifiants. On peut donc avoir une variable avec le même nom qu'un champ sans problème d'ambiguîté.

Structures "récursives". Une structure ne peut se contenir elle-même, sous peine d'engendrer une récursion infinie, donc pas de :

```
struct recurse {
  int truc;
  char machin;
  struct recurse bidule; // impossible, récursion infinie
};
```

Par contre une structure peut tout à fait contenir un pointeur vers une variable du même type structure (ou d'un autre type structure, évidemmment), ce qui est nécessaire pour implanter des collections d'objets dits "chaînés" (listes chaînées, arbres, certaines représentations de graphes, etc).

Exemple de déclaration d'une liste chaînée simple d'entiers :

```
struct maillon {
```

```
  int valeur; // valeur à mémoriser dans ce maillon de la liste
  struct maillon * suivant; // pointeur vers le maillon suivant
};
```

2 Définition de nom de types

Le mécanisme présenté ci-dessus présente le léger défaut d'introduire un nom composé pour les types structures, ainsi dans l'exemple précédent le nom de type est `struct client` et pas simplement `client` comme on aurait pu s'y attendre. D'une manière générale, il est de toute façon utile de pouvoir nommer des types de données à son gré, pour faciliter la compréhension du code. Ainsi une application manipulant des données relatives à un réseau électrique pourrait utiliser des types comme "voltage", "intensite" et "puissance". L'utilisation de noms clairs, permettrait notamment de détecter plus facilement des erreurs de programmation comme par exemple ajouter une intensité à un voltage. Le langage C supporte cette fonctionnalité en permettant la création de noms de types, sans toutefois avoir un contrôle de types strict à la ADA ou à la C++, ce qui en limite l'efficacité : ce n'est pas le compilateur qui vérifiera que vous n'additionnez pas des intensités à des voltages.

Un nouveau type se construit toujours à partir des types pré-existants, éventuellement composés en tableau, en structure ou union. La définition du nouveau type s'écrit comme une déclaration de variable de ce type précédée du mot-clé `typedef`. Le nom qui servirait de nom de variable sans `typedef`, devient un nom de type. Par convention, on écrit souvent en majuscules ces noms de types, pour les distinguer des noms de variables :

```
typedef description_du_type NOM_DU_TYPE;
```

Exemples :

```
typedef double INTENSITE;
typedef double VOLTAGE;
// INTENSITE et VOLTAGE sont des synonymes de double

VOLTAGE v1, v2; // deux variables de type VOLTAGE


typedef char CHAINE[80];

CHAINE ch1; // ch1 est un tableau de 80 caractères

typedef struct client {
  char nom[25];  // 1er champ
  int numero;    // 2ème champ
  double compte; // 3ème champ
} CLIENT;            // CLIENT est un synonyme du type "struct client"

CLIENT cl1, cl2;     // cl1 et cl2 sont de type CLIENT
```

A noter que le nouveau nom de type n'est disponible qu'à partir de l'instruction suivante, notamment lors de l'usage de structures contenant des pointeurs sur elles-mêmes :

```
struct maillon {
  int valeur;
  struct maillon * suivant; // et non pas MAILLON, encore inconnu ici
} MAILLON; // nouveau nom de type
```

IX Modularité et portée des variables

1 Modularité

De même que le découpage d'un programme en plusieurs fonctions sert à en améliorer la lisibilité, il est aussi souvent utile de découper une application en plusieurs fichiers qui isolent des fonctionnalités relativement indépendépendantes : on parle de modularité. D'une manière générale, un module est donc un ensemble de fonctionnalités que l'on veut rassembler dans un fichier, et qui ne constitue pas un programme complet (en particulier il n'y a pas de fonction `main`). L'idée est de compiler le module, et de pouvoir ré-utiliser ces fonctionnalités dans d'autres programmes, sans avoir à les réécrire. Les modules constituent donc des bibliothèques de fonctionnalités.

Le module peut donc mettre donc à disposition des programmes "clients" différents "objets" C : types, variables, fonctions. Pour réaliser certaines de ses fonctions il peut aussi utiliser des types, variables et fonctions

qui resteront privées, inaccessibles de l'extérieur du module. En C, le support de la modularité est limité au découpage en fichiers, à la possibilité de compiler ces fichiers séparèment et à une gestion minimaliste des noms des variables, types et fonctions utilisés dans ces fichiers pour les rendres visibles de l'extérieur ou, au contraire, privés.

2 Organisation d'un module

Idéalement un module se présente générallement en deux fichiers : l'en-tête (*header*) d'extension `.h`, et le corps du module d'extension `.c`. Dans le cas où le module offre seulement des types à partager, il n'y a pas besoin de fichier corps.

En-tête de module. Dans l'en-tête on va regrouper tous les éléments qui doivent être visibles de l'extérieur : types proposés par le module, déclarations préliminaires des fonctions proposées, et éventuellement déclaration de variables partagées par le module. Les déclarations de variables seront précédées du mot-clé `extern`.

L'en-tête ne sera pas compilé, mais sera inclus dans les programmes "clients" du module, par une directive `#include`. Attention, un fichier d'en-tête peut se retrouver inclus plusieurs fois indirectement, par exemple on inclus deux modules A et B qui ont eux-mêmes inclus tous les deux le module C : C se retrouve inclus deux fois ce qui génère souvent des erreurs de compilation. Par conséquent il faut donner des directives de compilation conditionnelle afin de n'inclure le module qu'une seule fois (voir exemple plus bas). Ces directives ont besoin d'un identifiant pour le module : par convention nous utiliserons le nom du fichier d'en-tête en remplaçant l'extension `.h` par `_h`.

Corps du module. Le fichier corps est d'extension `.c` et ne doit pas contenir de fonction `main`. On y trouve les définitions complètes des fonctions (leur code), la déclaration des variables partagées (sans `extern` cette fois), et tous les objets privés au module : déclaration de types, variables et fonctions privées du module préfixées par le mot-clé `static`. Attention, une variable déclarée au niveau du fichier (hors de toute fonction) sans être préfixée par `static` est visible à l'extérieur du module, ce qui peut entraîner un conflit de nom si une autre variable de même nom existe dans un autre fichier ! Le fichier corps va aussi inclure le fichier d'en-tête du module, ce qui permettra au compilateur de vérifier la correspondance des déclarations péliminaires de fonction avec leur définition.

Le corps du fichier n'est pas inclus dans le programme "client", il est compilé séparèment en demandant au compilateur de ne pas construire un exécutable mais simplement un fichier dit "objet" d'extension `.o` lequel sera "lié" lors de la compilation du programme client pour obtenir un exécutable.

Compilation du corps de module : `gcc -c module.c`

On obtient un fichier `module.o`.

Programme client. Le programme "client" du module se contente d'inclure le fichier d'en-tête :

```
#include "module.h"
```

On le compile en ajoutant le fichier objet :

```
gcc -o client client.c module.o -Wall
```

3 Exemple de module

On traite ici un exemple de module fictif qui regroupe toutes les possibilités offertes avec des types, variables et fonctions partagés et d'autres privés. On appelle ce module `exmod`, avec donc un fichier d'entête `exmod.h` et un fichier corps `exmod.c`Noter que le nom du fichier en-tête est utilisé dans sa directive de compilation conditionnelle.

Fichier d'en-tête : fichier `exmod.h`.

```
// directives de compilation conditionnelle pour inclure une seule fois le module
#ifndef exmod_h
#define exmod_h

// type partagé
typedef struct {
int a, b;
} EXSTRUCT;

// variable partagée
extern int a;
```

```
//déclaration préliminaire de fonction (extern n'est pas obligatoire pour
// les fonctions)
int truc(int x);

// directive de fin de compilation conditionnelle
#endif
```

Fichier corps : fichier `exmod.c`.

```
// inclure l'entête du module pour vérification
#include "exmod.h"

// type privé
typedef struct {
float x, y;
} STRUCTPRIVEE;

// variable partagée (sans extern ici !)
int a;

// variable privée
static int b;

//définition fonction
int truc(int x) {
  return toto(x);
}

// fonction privée
static int toto(int n) {
  return 2*n;
}
```

4 Portée des variables

Comme vu dans la section II, une variable peut être masquée temporairement par une autre de même nom dans un bloc imbriqué. Elle n'est alors plus accessible par son nom, bien qu'elle continue d'exister. On appelle "portée" l'étendue (ou les étendues, pas forcément contigües) du code où une variable est accessible : elle a été déclarée, et elle n'est pas masquée.

Principales règles en matière de portée des variables :

- la portée d'une variable ne dépend pas de l'exécution du code, elle est entièrement déterminée statiquement, lors de la compilation (donc il suffit de lire le code pour la déterminer).
- un nom de variable désigne la variable déclarée sous ce nom dans le bloc englobant le plus proche, à défaut de la trouver dans un bloc englobant, elle doit être déclarée au niveau du fichier ;
- une déclaration au niveau du fichier (hors de tout bloc), préfixée par `extern` désigne une variable déclarée au niveau fichier dans un autre module ;
- `extern` dans une fonction indique simplement que la variable est déclarée en dehors de la fonction :elle peut l'être dans le même module ;
- les paramètres formels d'une fonction se comportent comme des variables locales ;

Attention, C a des règles par défaut dangereuses (ex : une variable non déclarée est considérée comme de type int). Les règles précédentes sont pour une part des conventions de bonnes programmation. L'essentiel est illustré dans l'exemple suivant, présentant deux modules, m1 et m2, d'une application fictive :

module : m1.c

```
extern int i; // i vient de m2
int j;        // j global à m1 et m2
double f();   // f() n'est pas définie
// => il faut qu'elle le soit dans m2
// c'est donc une déclaration "extern"

static int g() { // g() est privée à m1
  // il masque dans m1 le g() global de m2
  return 1;
}

int h(double x) { // h globale à m1 et m2
  double y;
  extern int i,j; // externes à la fonc-
  // tion (i de m2, j de m1)

  y = f(x); // f() de m2,
  // x le paramètre passé à h()

  return g(); // g() de m1
}
```

module : m2.c

```
int i; // global à m1 et m2
static int j; // local à m2,
// ce  j masque dans m2 le j global de m1

double f() {  // définition de f()
  // f() globale à m1 et m2
  return 2.3;
}
int g() { // global à l'application
  // mais masqué dans m1
  extern int j; // celui de m2

  return (int) f() + j;
}
```

Notez dans l'exemple que le même nom de fonction `g()` et le même nom de variable `j` sont utilisés dans les deux modules. C'est possible grâce au mot-clé `static` qui restreint un nom à être privé à un module, et donc permet l'emploi du même nom dans d'autres modules. Sans `static`, il y aurait conflit de nom et on ne pourrait pas lier les deux modules dans une application. Cette technique est très utile : quand on écrit un module on ne sait pas forcément à l'avance avec quels autres modules il va être compilé, par conséquent tout ce qui peut être mis en privé avec `static` doit l'être, afin d'éviter de possibles futurs conflits de noms.

X Utilitaires

Un certain nombre de biblithèques (ou librairies — de l'anglais *library*) livrées en standard avec le compilateur étendent les capacités du langage de base en fournissant des fonctions utilitaires et accessoirement des types et des variables. Cette section présente une sélection de bibliothèques et de fonctionnalités.

1 Mode d'emploi

Pour utiliser une bibliothèque, il faut inclure son fichier d'en-tête :
`#include <entete.h>`
Dans la suite de la section les fonctions sont présentées en donnant leur déclaration préliminaire (type de valeur retournée, nom de fonction, type et nombre des paramètres). Pour utiliser les fonctions il faut donc transformer cette déclaration en un appel de fonction.

2 Entrées/sorties standard : stdio.h

FILE* fopen(const char* filename, const char* mode) ;
ouvre un fichier de nom filename et retourne un pointeur de fichier ou le pointeur nul si échec. Le mode d'ouverture est précisé par une chaîne :
- "r" : lecture
- "w" : écriture (le fichier est effacé s'il existe)
- "a" : écriture avec positionnement pour ajout en fin
- "r+" : lecture + écriture
- "w+" : lecture + écriture (le fichier est effacé s'il existe)
- "a+" : lecture + écriture avec positionnement pour ajout en fin

int fflush(FILE* stream) ;
force la copie des données sur stream (ouvert en écriture) en vidant les tampons provisoires et retourne 0 si succès, EOF si erreur. fflush(0) vide tous les tampons actuellement ouverts en écriture.

int fclose(FILE* stream) ;

ferme le fichier (après avoir fait un flush si en écriture) et retourne 0 si succès, EOF si erreur.

int fprintf(FILE* stream, const char* format, ...) ;

Convertit les données (selon les indications de la chaîne format) et les envoie sur le flux de sortie stream, et retourne le nombre de caractères sortis, ou une valeur négative si erreur. Un format de conversion consiste en :

- un caractère `%` suivi de :
- un caractère optionnel parmi :

  - sortie alignée à gauche

  \+ sortie alignée à droite

  espace le + d'un nombre positif est remplacé par un espace

  0 la sortie est complétée à gauche par des 0
- nombre optionnel indiquant la largeur minimale de sortie : si specifié par `*`, la valeur est donnée pae le prochain argument non utilisé.
- point optionel `.` qui précède la précision demandée (voir après).
- précision optionnelle : si le caractère de conversion est `s`, nombre max de caractère de la chaîne à sortir, si le carcatère est dans {eEf}, nombre de chiffres après le point décimal, pour {gG}, chiffres significatifs, pour un entier, nombre minimum de chiffres à sortir. Si donné par *, la valeur est donnée pae le prochain argument non utilisé.
- caractère optionnel modificateur de type :

  h short

  l long

  L long double
- caractère de conversion du prochain argument

  d,i int, en décimal

  o int, en octal

  x,X int, en héxadécimal

  u int, en décimal non signé

  c char

  s char* (chaîne)

  f double, au format [-]mmm.ddd

  e,E double, au format [-]m.dddddd(e|E)(+|-)xx

  g,G double

  p void*, comme une adresse numérique

  % sort un signe %

int printf(const char* format, ...) ;

équivalent de `fprintf(stdout, f, ...)` : sort sur la sortie standard (écran).

int sprintf(char* s, const char* format, ...) ;

comme `fprintf`, mais la sortie est envoyée dans une chaîne. Le nombre de caractères (hors `'\0'`) est retourné.

int fscanf(FILE* stream, const char* format, ...) ;

Effectue une lecture avec conversion, lisant dans `stream` selon les indications de `format`. Retourne le nombre d'arguments remplis (lus), ou `EOF` si fin de fichier ou erreur avant toute conversion. Chacun des arguments qui suit le format, doit êtreun pointeur. Un format de conversion consiste en :

- des espaces ou tabulations, qui seront sautés
- des caractères ordinaires qui doivent correspondre à ceux du flux d'entrée et qui seront ignorés (à éviter !)
- une spécification de conversion débutée par le caractère `%`
- `*` (optionnel) qui indique de passer le prochain argument
- une largeur de champ de lecture, optionnelle
- caractère optionnel modificateur de type :

  h short

  l long

  L long double

- caractère de conversion pour le prochain argument

  d argument int* requis, en décimal

  i int* requis, décimal, octal ou hexa

  o int* requis, en octal

  x int* requis, en héxadécimal

  u unsigned int* requis, en décimal non signé

  c char * requis ; les espace ne sont pas sautés

  s char* (chaîne terminée par 0) ; les espaces stoppent la saisie,

  e,f,g float*, valeur flottante

int scanf(const char* format, ...) ;
Equivalent à `fscanf(stdin, f, ...)` : lit depuis l'entrée standard (clavier).

int sscanf(char* s, const char* format, ...) ;
comme `fscanf()`, mais lit depuis la chaîne `s`.

int fgetc(FILE* stream) ;
Retourne le prochain caractère de `stream`, ou `EOF` si fin de fichier ou erreur.

char* fgets(char* s, int n, FILE* stream) ;
Lit des caractères depuis`stream` pour les copier dans `s`, s'arrête quand `n-1` caractères sont copiés, ou si un `'\n'` est copié, ou si fin de fichier ou erreur. En l'absence d'erreur, `s` est une chaîne correcte. Retourne 0 si fin de fichier ou erreur, ou sinon `s`.

int fputc(int c, FILE* stream) ;
Ecrit `c` dans `stream`. Retourne c, ou `EOF` si erreur.

char* fputs(const char* s, FILE* stream) ;
Ecrit `s` dans `stream`. Retourne non-negatif si succès, ou `EOF` si erreur.

int ungetc(int c, FILE* stream) ;
Simule le "renvoi" de `c` (qui doit être un caractère), dans le flux d'entrée `stream` de telle sorte qu'il sera retourné par la prochaine lecture. Un seul caractère peut être ainsi renvoyé. Retourne `c`, ou `EOF` si erreur.

int fread(void* ptr, int size, int nobj, FILE* stream) ;
Lit (au plus) `nobj` objets de taille `size` depuis le fichier associé à `stream` et les copies en mémoire à l'adresse `ptr`. Retourne le nombre d'objets lus (`feof` et `ferror` peuvent être testés). Note : les `int` peuvent être longs.

int fwrite(const void* ptr, int size, int nobj, FILE* stream) ;
Ecrit dans `stream`, `nobj` objets de taille `size` copiés depuis le tableau `ptr`. Retourne le nombre d'objets écrits. Note : les `int` peuvent être longs.

int fseek(FILE* stream, long offset, int origin) ;
Positionne le fichier associé au flux `stream` sur la position `origin` + `offset`. Trois valeurs sont possibles pour `origin` :
- début de fichier pour `SEEK_SET`,
- position courante pour `SEEK_CUR`,
- fin de fichier pour `SEEK_END`.

Retourne non-zero si erreur.

long ftell(FILE* stream) ;
Retourne la position courante dans le fichier associé au flux `stream`, ou -1 si erreur.

void rewind(FILE* stream) ;
Equivalent à fseek(stream, 0L, `SEEK_SET`) ;

int feof(FILE* stream) ;
Retourne vrai (c'est à dire !=0) si l'indicateur "fin de fichier" a été positionné pour le flux `stream` (il faut avoir tenté une lecture en fin de fichier pour qu'il le soit).

int ferror(FILE* stream) ;
Retourne non-zero si l'indicateur d'erreur est positionné pour `stream`.

3 Traitement de chaînes : string.h

int strlen(const char* cs) ;
Retourne la longueur de la chaîne `cs`.

char* strcpy(char* s, const char* ct) ;
Copie `ct` dans `s` (et retourne `s`). Attention ! pas de test si s est assez grand pour contenir ct !

char* strncpy(char* s, const char* ct, int n) ;
Copie au plus `n` caractères de ct dans s, complète s avec 0 si possible (et retourne `s`). Attention à vous d'assurer que s peut contenir s+1 caractères, pour le marqueur de fin.

char* strcat(char* s, const char* ct) ;
Concatène ct à s (et retourne `s`). Attention ! pas de test si s est assez grand pour contenir ct !

char* strncat(char* s, const char* ct, int n) ;
Concatène au plus n caractères de ct à s (et retourne `s`). Attention ! pas de test si s est assez grand pour contenir ct !

int strcmp(const char* cs, const char* ct) ;
Compare cs avec ct, selon l'ordre du jeu de caractèreres utilisé. Retourne une valeur entière à tester :
- negative value si cs<ct (cs avant ct),
- zéro si cs est égal à ct,
- positive si cs>ct (cs après ct).

int strncmp(const char* cs, const char* ct, int n) ;
Compare les n premiers caractères au plus de cs et ct. Même sémantique que `strcmp()`.

void* memcpy(void* s, const void* ct, int n) ;
Copie brute de mémoire : copie n octets de l'adresse ct vers l'adresse s and retourne s. Risque d'erreurs si l'intersection zone de destination / zone origine n'est pas vide.

4 Nombres aléatoires : stdlib.h

Attention le générateur aléatoire standard du C ne possède pas de très bonnes propriétés statistiques, il est recommandé d'utiliser une librairie spécialisée (plusieurs exemples comme SVID sont disponibles sur internet) pour les utilisations sérieuses.

void srand(unsigned int seed) ;
Initialise le générateur pseudo-aléatoire, ce qui doit typiquement être fait une seule fois dans le programme, à son démarrage. Le paramètre `seed` est la graine du générateur : elle lui donne son point de départ. Lorque deux exécutions utilisent la même graine, les nombre générés seront les mêmes, ce qui peut être utile pour mettre au point le programme. Une fois la mise au point terminée, on utilise souvent comme graine l'heure obtenue par l'appel à la fonction `times(0)`.

int rand(void) ;
Retourne un entier pseudo-aléatoire dans l'intervalle $[0, RAND_MAX]$. On peut typiquement convertir cet entier en flottant dans l'intervalle $[0, 1[$ par l'expression : `rand()/(1.0 + (double) RAND_MAX)`.

XI Extras

On présente ici quelques constructions peu fréquemment utilisées

1 Unions

Il arrive qu'on souhaite pouvoir stocker dans une variable "struct" des données telles que leur "format" (nombre, taille et type de données élémentaires) dépend lui-même de la donnée à stocker (on parle d'enregistrement avec partie variable). Par exemple un logiciel de dessin peut mémoriser des formes géométriques dessinées par l'utilisateur : dans le cas d'un carré, on peut le stocker par les 2 paires de coordonnées (coin en haut à gauche et coin en bas à droite), mais pour un cercle il faut fournir la coordonnée du centre et son rayon... On pourrait alors obtenir un gain de place en utilisant la même zone de stockage mémoire soit pour la 2ème paire de coordonnées, soit pour le rayon, selon le type de figure. C'est ce qui est rendu possible avec une variante de "struct" appelée "union" :

```
struct coord {
  int x, y; // une paire de coordonnées à l'écran
};

struct figure {
  int genre; // 0 = carré, 1 = cercle
  struct coord c1; // 1ère coord : point haut gauche si carré, centre si cercle
  union extra{
```

```
    struct coord c2; // 2nd coord si carré
    int rayon; // rayon si cercle
  } extra; // données complémentaires
};

int main() {
  struct figure f;

  // utilisation de f comme un carré
  f.genre = 0;
  f.c1.x = 10;
  f.c1.y = 50;
  f.extra.c2.x = 40;
  f.extra.c2.y = 120;

  // on change d'avis, f doit stocker un cercle
  f.genre = 1;
  f.extra.rayon = 15; // les données stockées dans f.extra.c2 sont écrasées
}
```

Dans cet exemple il faut comprendre que la structure utilise le même espace mémoire (celui de `union extra`) pour ranger soit la paire de coordonnées `c2` soit la donnée `rayon`. Donc en aucun cas on ne peut mémoriser les deux en même temps, c'est pourquoi il est généralement indispensable d'ajouter un champ qui mémorise quelle est la configuration qui a actuellement cours : c'est le rôle du champ `genre` dans l'exemple. On peut éventuellement se dispenser de ce champ, à ses risques et périls : notez que le langage C ne contrôle pas la cohérence des accès aux champs d'une union (on pourrait écrire dans `extra.c2` et relire dans `extra.rayon`!)... Nous ne détaillons pas davantage l'utilisation des unions, qui sont réservées à des utilisations très particulières et qui sont rendus grandement obsolètes dans les "langages à objets" comme C++.

2 Variables "statiques" rémanentes

On a vu que le mot-clé `static` servait à indiquer qu'une variable ou fonction est privée à un module. Les concepteurs de C ont maladroitement utilisé le même mot-clé pour un autre concept, celui de variable "rémanente" :

Une variable `static` déclarée dans une fonction est certes privée à cette fonction (elle l'aurait été même sans `static`), mais de plus elle est allouée dans la zone statique de la mémoire, comme si elle avait été déclarée au niveau du fichier. N'étant pas détruite à la fin du bloc fonction, elle garde sa valeur entre deux appels de la fonction, contrairement au variables locales habituelles! Si elle est initialisée lors de sa déclaration, cette initialisation est effectuée une seule fois, en début de programme. Cette astuce permet par exemple de compter combien de fois une fonction est appelée dans un programme.

```
void truc() {
  static int cpt = 0; // variable rémanente
    // initialisée à 0 une seule fois, en début de programme

  cpt += 1; // exécuté à chaque appel
  printf("truc() vient d'être appelée pour la %d ème fois\n", cpt);
}
```

3 Mots-clés `const` et `volatile`

Les mot-clés `const` et `volatile` sont des "qualificateurs de types" introduits dans le C90 : ils sont placés en préfixe d'un type ou d'une variable pour préciser le comportement du compilateur.

Mot-clé `const`. Les variables du type préfixé par `const` sont considérées comme constantes : le compilateur interdira les affectations à ces variables. Par contre on peut (et on devrait) les initialiser à leur création : c'est a priori la seule façon de leur attribuer une valeur, quoiqu'il soit aussi possible d'utiliser une conversion dans un type non constant mais dans ce dernier cas on peut légitiment se demander l'intérêt de lever une interdiction qu'on a soi même posée.

Au contraire d'une constante du pré-processeur (voir section 2.vi) qui sont de simples "copiés-collés" de littéraux, les constantes définie avec `const` sont effectivement *stockées* en mémoire comme les variables : elles

sont donc typées et possèdent une adresse. Pour que la protection contre l'écriture soit effective, il faut donc qu'elle s'étende aussi à l'éventualité de modifier la constante indirectement par son adresse. On ne pourra donc affecter l'adresse d'une constante `const` que dans un pointeur sur un objet constant (attention, gcc ne fait que signaler un "warning" et pas une erreur si on affecte dans un pointeur ordinaire). Le fait que le pointeur lui-même puisse ou pas être constant est complètement indépendant.

Exemples :

```
const int a = 12; // a est une constante entière
int const aa = 12; // autre écriture possible aa est une constante entière
int b = 5;

printf("\%d\n", a); // ok : lecture
a = 3; // INTERDIT : écriture

int *p1 = &a; // WARNING : p1 pointe une variable entière, pas une constante

const int *p2 = &a; // ok, p2 est un pointeur sur constante entière
int const *pp2 = &a; // ok, autre écriture possible: pp2 est comme p2
*p2 = 3; // INTERDIT : p2 est un pointeur de constante

p2 = &b; // ok, même si b n'est pas une constante
*p2 = 6; // INTERDIT car p2 est un pointeur sur constante (bien que b non constant)


int * const p3 = &b; // notez la place de const juste avant p3
*p3 = 6; // pas de problème c'est l'adresse qui est constante, pas le contenu
p3 = &b; // interdit : l'adresse dans p3 est constante
```

Mot-clé `volatile`. Rarement utilisé, le qualificateur `volatile` indique au compilateur qu'il ne doit pas optimiser la variable en question. En particulier le compilateur n'utilisera pas de copie de la valeur de la variable même si elle ne semble pas avoir été modifiée depuis l'obtention de la copie.

Il est généralement utilisé lorsqu'une variable est mise à jour par un autre processus que le programme (adresse associée à un périphérique matériel, ou partagée entre plusieurs processus), auquel cas le compilateur ne pouvant pas savoir quand la variable est effectivement modifiée, il ne doit pas chercher à optimiser son traitement.

4 Mot-clé `enum`

Le mot-clé `enum` ne définit pas réellement un type énuméré à la ADA, mais crée seulement des littéraux d'une manière proche de `#define`.

```
enum LETTRES{
      A, B, C, D='D', E, F, G=48, H
} v1;
```

La variable v1 est de type entier de même que tous les symboles A à H, qui ont chacun une valeur entière : A est par défaut mis à 0, les suivant prennent les valeurs entières suivantes, donc B vaut 1, et C vaut 2. On peut fixer aussi une valeur, comme ici D qui vaut la valeur du caractère 'D' dans le code de caractères utilisé. E et F prendront alors les valeurs suivantes donc respectivement celles des caractères 'E' et 'F'. De même G prend la valeur 48 et H == 49.

On peut par exemple utiliser ces symboles (sans apostrophes ni guillemets, ce ne sont ni des chaînes ni des caractères) dans une structure `switch` pour améliorer la lisibilité, tout comme on pourrait le faire avec des constantes littérales du pré-processeur.

Par contre la portée de ces symboles est celle de leur déclaration, et pas forcément celle du fichier comme pour les constantes du pré-processeur.

5 Pointeurs de fonctions

Le code d'une fonction étant stocké en mémoire, il a une adresse. On peut récupérer cette adresse et réaliser un appel de fonction.

Pour récupérer l'adresse il faut déclarer un pointeur de fonction : on donne le type de retour de la fonction, le nom du pointeur précédé d'une étoile comme d'habitude mais le tout entre parenthèses, suivi entre parenthèses

de la liste des types des paramètres attendus (liste vide mais parenthèses obligatoires si c'est une fonction sans paramètres).

Lors de l'appel on dépointe le pointeur, toujours entre parenthèses, et on passe les paramètres comme pour une fonction normale.

Si plusieurs fonctions sont de même type, on peut stocker leurs adresses dans un tableau comme dans l'exemple (à compiler avec la librairie mathématique : `gcc -o test test.c -lm`) :

```
#include <stdio.h>
void bonjour() {  printf("Bonjour\n"); } // une fonction

extern double fabs(double); // 3 membres de la librairie math.h
extern double sin(double);
extern double cos(double);

int main() {
  // declaration et initialisation des pointeurs de fonctions
  void (* pf)() = &bonjour;
  double (*funtab[3])(double) = {&fabs,&sin,&cos};

  // utilisations des pointeurs
  (*bonjour)(); // affiche message de bienvenue
  int a; float x;
  printf("entrez un entier entre 0 et 2 et un flottant:");
  scanf("%d %f", &a, &x);
  printf("%f\n", (*(funtab[a]))(x)); // affiche la valeur de la fonction choisie

  return 0;
}
```

Le même principe permet à la librairie `stdlib.h` d'offrir une fonction de tri générique :

void *qsort(const void *base, int nmemb, int size, int (*compar)(const void *, const void *));

Tri par l'algorithme du "quick sort", du tableau d'adresse `base`, de taille `nmemb` éléments, chaque élément étant lui même de taille `size` (obtenu par l'opérateur `sizeof`). Le tri nécessite qu'on lui passe l'adresse d'une fonction de comparaison permettant d'obtenir l'ordre relatif entre deux éléments du tableau : cette fonction prend deux pointeurs sur les objets à comparer, et renvoie -1, 0 ou 1 selon que le premier est avant, égal à ou après le second. Les pointeurs paramètres de la fonction de comparaison sont génériques (`void*`), car ils sont passés par `qsort()` qui ne peut pas connaître le type qui va lui être soumis. A l'intérieur de votre fonction de comparaison il suffit de convertir les pointeurs génériques en pointeurs typés, afin de pouvoir accéder aux éléments du tableau.

```
int triEntierDecroissant(const void *p1, const void *p1) {
  // ma fonction de tri, pour des entiers, donne l'ordre décroissant
  const int *e1 = ( const int *) p1;
  const int *e2 = ( const int *) p2;

  if (*e1 > *e2)
    return -1; // *e1 et *e2 sont deja dans l'ordre decroissant
  if (*e1 < *e2)
    return 1;  // *e1 doit être apres *e2
  else
    return 0; // egalite
} // triEntierDecroissant


int main() {
  int tab[5]={12, 3 ,5 , 7, 4};

  // appel du tri avec ma fonction de comparaison de 2 elements du tableau
  qsort((void *) tab, 5, sizeof(int), &triEntierDecroissant);

  // affichage
```

```
  int i;
  for (i = 0; i < 5; i++)
    printf("%d\n", tab[i]);

  return 0;
}
```